exec manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/exec.htm

NAME

exec — Invoke subprocesses

SYNOPSIS

exec ?*switches*? *arg* ?*arg ...*? ?**&**?

DESCRIPTION

This command treats its arguments as the specification of one or more subprocesses to execute. The arguments take the form of a standard shell pipeline where each *arg* becomes one word of a command, and each distinct command becomes a subprocess. The result of the command is the standard output of the final subprocess in the pipeline, interpreted using the system **encoding**; to use any other encoding (especially including binary data), the pipeline must be **open**ed, configured and read explicitly.
If the initial arguments to **exec** start with **-** then they are treated as command-line switches and are not part of the pipeline specification. The following switches are currently supported:

-ignorestderr
Stops the **exec** command from treating the output of messages to the pipeline's standard error channel as an error case.

-keepnewline
Retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.

--
Marks the end of switches. The argument following this one will be treated as the first *arg* even if it starts with a **-**.

If an *arg* (or pair of *arg*s) has one of the forms described below then it is used by **exec** to control the flow of input and output among the subprocess(es). Such arguments will not be passed to the subprocess(es). In forms such as "**<** *fileName*", *fileName* may either be in a separate argument from "**<**" or in the same argument with no intervening space (i.e. "**<***fileName*").

|
Separates distinct commands in the pipeline. The standard output of the preceding command will be piped into the standard input of the next command.

|&

Separates distinct commands in the pipeline. Both standard output and standard error of the preceding command will be piped into the standard input of the next command. This form of redirection overrides forms such as 2> and >&.

< fileName
The file named by *fileName* is opened and used as the standard input for the first command in the pipeline.

<@ fileId
FileId must be the identifier for an open file, such as the return value from a previous call to **open**. It is used as the standard input for the first command in the pipeline. *FileId* must have been opened for reading.

<< value
Value is passed to the first command as its standard input.

> fileName
Standard output from the last command is redirected to the file named *fileName*, overwriting its previous contents.

2> fileName
Standard error from all commands in the pipeline is redirected to the file named *fileName*, overwriting its previous contents.

>& fileName
Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, overwriting its previous contents.

>> fileName
Standard output from the last command is redirected to the file named *fileName*, appending to it rather than overwriting it.

2>> fileName
Standard error from all commands in the pipeline is redirected to the file named *fileName*, appending to it rather than overwriting it.

>>& fileName
Both standard output from the last command and standard error from all commands are redirected to the file named *fileName*, appending to it rather than overwriting it.

>@ fileId
FileId must be the identifier for an open file, such as the return value from a previous call to **open**. Standard output from the last command is redirected to *fileId*'s file, which must have been opened for writing.

2>@ fileId
FileId must be the identifier for an open file, such as the return value from a previous call to **open**. Standard error from all commands in the pipeline is redirected to *fileId*'s file. The file must have been opened for writing.

2>@1

Standard error from all commands in the pipeline is redirected to the command result. This operator is only valid at the end of the command pipeline.

>&@ *fileId*
FileId must be the identifier for an open file, such as the return value from a previous call to **open**. Both standard output from the last command and standard error from all commands are redirected to *fileId*'s file. The file must have been opened for writing.

If standard output has not been redirected then the **exec** command returns the standard output from the last command in the pipeline, unless “2>@1” was specified, in which case standard error is included as well. If any of the commands in the pipeline exit abnormally or are killed or suspended, then **exec** will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the **-errorcode** return option will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file and that standard error is not redirected and **-ignorestderr** is not specified, then **exec** will return an error; the error message will include the pipeline's standard output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is normally deleted from the result or error message. This is consistent with other Tcl return values, which do not normally end with newlines. However, if **-keepnewline** is specified then the trailing newline is retained.

If standard input is not redirected with “<”, “<<” or “<@” then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *arg* is “&” then the pipeline will be executed in background. In this case the **exec** command will return a list whose elements are the process identifiers for all of the subprocesses in the pipeline. The standard output from the last command in the pipeline will go to the application's standard output if it has not been redirected, and error output from all of the commands in the pipeline will go to the application's standard error file unless redirected.

The first word in each command is taken as the command name; tilde-substitution is performed on it, and if the result contains no slashes then the directories in the PATH environment variable are searched for an executable by the given name. If the name contains a slash then it must refer to an executable reachable from the current directory. No “glob” expansion or other shell-like substitutions are performed on the arguments to commands.

PORTABILITY ISSUES

Windows (all versions)
Reading from or writing to a socket, using the “**@** *fileId*” notation, does not work. When reading from a socket, a 16-bit DOS application will hang and a 32-bit application will

return immediately with end-of-file. When either type of application writes to a socket, the information is instead sent to the console, if one is present, or is discarded.
Note that the current escape resp. quoting of arguments for windows works only with executables using CommandLineToArgv, CRT-library or similar, as well as with the windows batch files (excepting the newline, see below). Although it is the common escape algorithm, but, in fact, the way how the executable parses the command-line (resp. splits it into single arguments) is decisive.

Unfortunately, there is currently no way to supply newline character within an argument to the batch files (**.cmd** or **.bat**) or to the command processor (**cmd.exe /c**), because this causes truncation of command-line (also the argument chain) on the first newline character. But it works properly with an executable (using CommandLineToArgv, etc).

Argument quoting

The arguments of the **exec** command are mapped to the arguments of the called program. Additional quote characters (**"**) are automatically added around arguments if expected. Special characters are escaped by inserting backslash characters.
The MS-Windows environment does execute programs mentioned in the arguments and called batch files (conspec) replace environment variables, which may have side effects (vulnerabilities) or break any already existing quoting (for example, if the environment variable contains a special character like a **"**). Examples are:

```
% exec my-echo.cmd {test&whoami}
  test
  mylogin
% exec my-echo.cmd "ENV X:%X%"
  ENV X: CONTENT OF X
```

The following formatting is automatically performed on any argument item to avoid subprogram execution: Any special character argument containing a special character (**&**, **|**, **^**, **<**, **>**, **!**, **(**, **)**, **(**, **%**) is automatically enclosed in quotes (**"**). Any data quote is escaped by insertion of backslash characters.
The automatic resolving of environment variables using "**%var%**" is critical, but has more use than danger and is not escaped.

TCL 8.6.10 refined this quoting by adding quoting for data quotes and individual quoting of "**%**". This may break present scripts which rely on the replacement functionality of environment variables. Thus, the individual quoting of "**%**" was removed in TCL 8.6.14, as environment variables are seen more helpful than a problem. A solution with command parameters is envisaged for a future release of TCL.

The Tk console text widget does not provide real standard IO capabilities. Under Tk, when redirecting from standard input, all applications will see an immediate end-of-file; information redirected to standard output or standard error will be discarded.

Either forward or backward slashes are accepted as path separators for arguments to Tcl commands. When executing an application, the path name specified for the application may also contain forward or backward slashes as path separators. Bear in mind, however, that most Windows applications accept arguments with forward slashes only as option delimiters and backslashes only in paths. Any arguments to an application that specify a path name with forward slashes will not automatically be converted to use the backslash character. If an argument contains forward slashes as the path separator, it may or may not be recognized as a path name, depending on the program.

Two or more forward or backward slashes in a row in a path refer to a network path. For example, a simple concatenation of the root directory **c:/** with a subdirectory **/windows/system** will yield **c://windows/system** (two slashes together), which refers to the mount point called **system** on the machine called **windows** (and the **c:/** is ignored), and is not equivalent to **c:/windows/system**, which describes a directory on the current computer. The **file join** command should be used to concatenate path components.

Note that there are two general types of Win32 console applications:

1. CLI — CommandLine Interface, simple stdio exchange. **netstat.exe** for example.
2. TUI — Textmode User Interface, any application that accesses the console API for doing such things as cursor movement, setting text color, detecting key presses and mouse movement, etc. An example would be **telnet.exe** from Windows 2000. These types of applications are not common in a windows environment, but do exist.

exec will not work well with TUI applications when a console is not present, as is done when launching applications under wish. It is desirable to have console applications hidden and detached. This is a designed-in limitation as **exec** wants to communicate over pipes. The Expect extension addresses this issue when communicating with a TUI application.

When attempting to execute an application, **exec** first searches for the name as it was specified. Then, in order, **.com**, **.exe**, **.bat** and **.cmd** are appended to the end of the specified name and it searches for the longer name. If a directory name was not specified as part of the application name, the following directories are automatically searched in order when attempting to locate the application:

- The directory from which the Tcl executable was loaded.
- The current directory.
- The Windows 32-bit system directory.
- The Windows home directory.
- The directories listed in the path.

In order to execute shell built-in commands like **dir** and **copy**, the caller must prepend the desired command with "**cmd.exe /c** " because built-in commands are not implemented using executables.

Unix (including Mac OS X)

The **exec** command is fully functional and works as described.

UNIX EXAMPLES

Here are some examples of the use of the **exec** command on Unix. To execute a simple program and get its result:

```
exec uname -a
```

WORKING WITH NON-ZERO RESULTS

To execute a program that can return a non-zero result, you should wrap the call to **exec** in **catch** and check the contents of the **-errorcode** return option if you have an error:

```
set status 0
if {[catch {exec grep foo bar.txt} results options]} {
    set details [dict get $options -errorcode]
    if {[lindex $details 0] eq "CHILDSTATUS"} {
        set status [lindex $details 2]
    } else {
        # Some other error; regenerate it to let caller handle
        return -options $options -level 0 $results
    }
}
```

This is more easily written using the **try** command, as that makes it simpler to trap specific types of errors. This is done using code like this:

```
try {
    set results [exec grep foo bar.txt]
    set status 0
} trap CHILDSTATUS {results options} {
    set status [lindex [dict get $options -errorcode] 2]
}
```

WORKING WITH QUOTED ARGUMENTS

When translating a command from a Unix shell invocation, care should be taken over the fact that single quote characters have no special significance to Tcl. Thus:

```
awk '{sum += $1} END {print sum}' numbers.list
```

would be translated into something like:

```
exec awk {{sum += $1} END {print sum}} numbers.list
```

WORKING WITH GLOBBING

If you are converting invocations involving shell globbing, you should remember that Tcl does not handle globbing or expand things into multiple arguments by default. Instead you should write things like this:

```
exec ls -l {*}[glob *.tcl]
```

WORKING WITH USER-SUPPLIED SHELL SCRIPT FRAGMENTS

One useful technique can be to expose to users of a script the ability to specify a fragment of shell script to execute that will have some data passed in on standard input that was produced by the Tcl program. This is a common technique for using the *lpr* program for printing. By far the simplest way of doing this is to pass the user's script to the user's shell for processing, as this avoids a lot of complexity with parsing other languages.

```
set lprScript [get from user...]
set postscriptData [generate somehow...]

exec $env(SHELL) -c $lprScript << $postscriptData
```

WINDOWS EXAMPLES

Here are some examples of the use of the **exec** command on Windows. To start an instance of *notepad* editing a file without waiting for the user to finish editing the file:

```
exec notepad myfile.txt &
```

To print a text file using *notepad*:

```
exec notepad /p myfile.txt
```

WORKING WITH CONSOLE PROGRAMS

If a program calls other programs, such as is common with compilers, then you may need to resort to batch files to hide the console windows that sometimes pop up:

```
exec cmp.bat somefile.c -o somefile
```

With the file *cmp.bat* looking something like:

```
@gcc %*
```

or like another variant using single parameters:

```
@gcc %1 %2 %3 %4 %5 %6 %7 %8 %9
```

WORKING WITH COMMAND BUILT-INS

Sometimes you need to be careful, as different programs may have the same name and be in the path. It can then happen that typing a command at the DOS prompt finds *a different program* than the same command run via **exec**. This is because of the (documented) differences in behaviour between **exec** and DOS batch files.
When in doubt, use the command **auto_execok**: it will return the complete path to the program as seen by the **exec** command. This applies especially when you want to run “internal” commands like *dir* from a Tcl script (if you just want to list filenames, use the **glob** command.) To do that, use this:

```
exec {*}[auto_execok dir] *.tcl
```

WORKING WITH NATIVE FILENAMES

Many programs on Windows require filename arguments to be passed in with backslashes as pathname separators. This is done with the help of the **file nativename** command. For example, to make a directory (on NTFS) encrypted so that only the current user can access it requires use of the *CIPHER* command, like this:

```
set secureDir "~/Desktop/Secure Directory"
file mkdir $secureDir
exec CIPHER /e /s:[file nativename $secureDir]
```